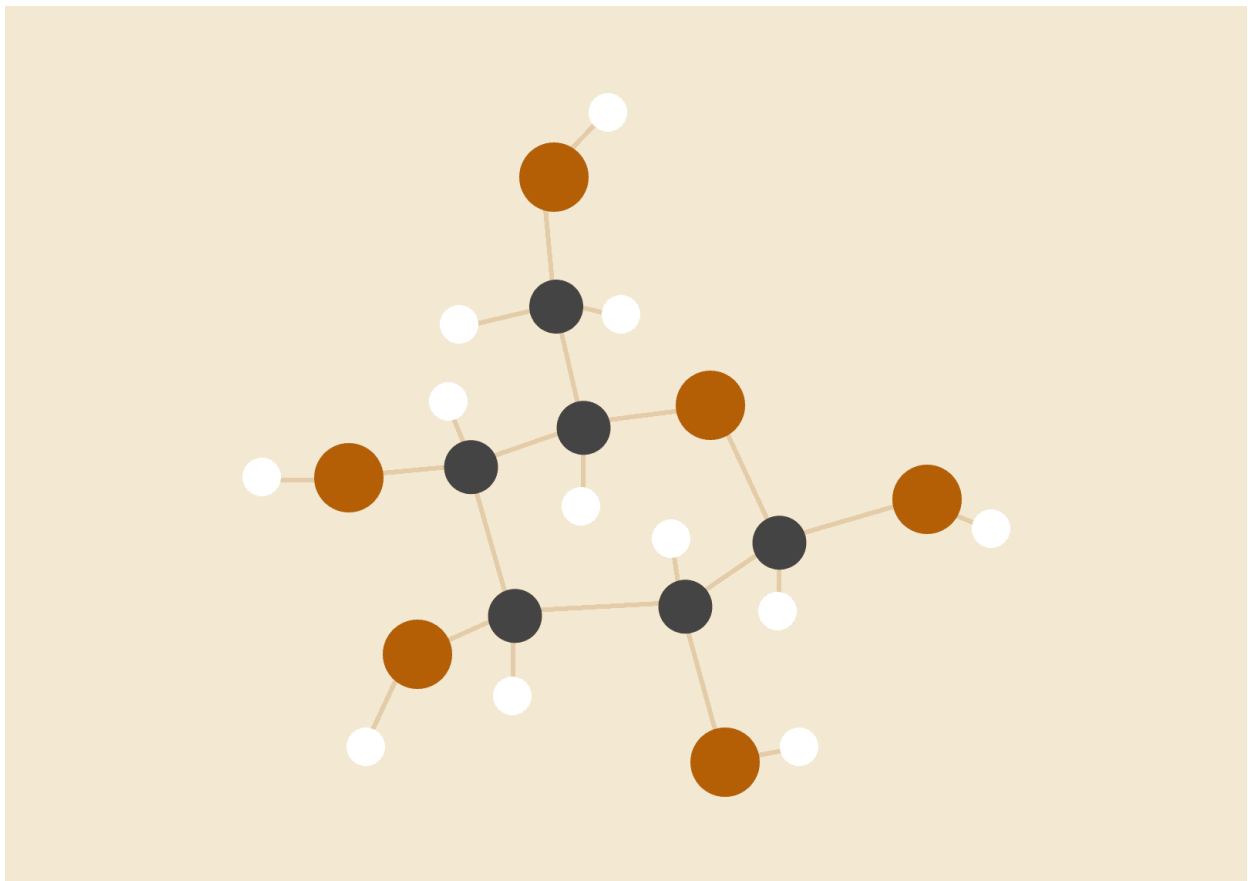


# ALGORITHM & DESIGN ANALYSIS

*Case Study: Dynamic vs Greedy Algorithms*



**Gurleen Singh Dhody**

**Sudhanshu Siddh**

11.30.2016  
MSCS Fall 2016

## INTRODUCTION

A divide and conquer algorithm partitions a problem into sub problems, solves the sub problems recursively and then combines their solutions to solve the original problem. But there are instances where the sub problems share sub-sub-problems, in such scenarios a divide and conquer does more work than necessary, solving the common sub problems repeatedly. In such scenarios, we resort to solving these problems using other alternative algorithms which also solve the problem by solving the sub problems but with a variation.

**Dynamic Programming:** Dynamic Programming also solves the problem by combining the solutions to the overlapping sub-problems. But dynamic programming solves each sub problem only once, thereby reducing the number of computations and then keeps a record of the solution for each sub-problem. Next time when a previously computed result is needed it is simply looked up through a  $O(1)$  complexity table.

**Greedy Algorithm Approach:** A greedy algorithm solves the problem by making the locally optimal choice at each stage with hope of finding a global optimal. A greedy algorithm makes a choice that seems best and then solves the sub problems that arise next. The choice made by greedy algorithm may depend on the choices made previously but doesn't depend on the future choices or all solutions to the sub-problems like in case of Dynamic Programming.

The main difference between greedy algorithms and dynamic programming is that instead of first finding optimal solutions to sub problems and then making informed choice, greedy algorithms first make a greedy choice, the choice that looks best at the time, and then solves a resulting sub problem, without bothering to solve all possible related smaller sub problems

## PROPOSAL

In this project, we intend to apply both algorithms on a set 6 problems statements and then crucially compare the performance of each algorithm for each problem statement. Thereby concluding which algorithm is a better choice. Parameters for comparing the algorithms:

- We will compare the algorithms by calculating the space ( $\Gamma$ ) and time ( $\theta$ ) complexity in each case. Graphs for the computations will be drawn accordingly where needed.
- We will also compare the algorithms based on the optimal solution provided for each problem statement.

This helps us determine which algorithm performs better for a specific problem.

## Case Study 1: Dungeon Game

### Problem Statement:

- The demons had captured the princess (P) and imprisoned her in the bottom-right corner of a dungeon. The dungeon consists of  $M \times N$  rooms laid out in a 2D grid. Our valiant knight (K) was initially positioned in the top-left room and must fight his way through the dungeon to rescue the princess.
- The knight has an initial health point represented by a positive integer. If at any point his health point drops to 0 or below, he dies immediately.
- Some of the rooms are guarded by demons, so the knight loses health (negative integers) upon entering these rooms; other rooms are either empty (0's) or contain magic orbs that increase the knight's health (positive integers).

In order to reach the princess as quickly as possible, the knight decides to move only rightward or downward in each step.

Write a function to determine the knight's minimum initial health so that he is able to rescue the princess.

*For example*, given the dungeon below, the initial health of the knight must be at least 7 if he follows the optimal path RIGHT-> RIGHT -> DOWN -> DOWN.

-2 (K)	-3	3
-5	-10	1
10	30	-5 (P)

Notes:

1. The knight's health has no upper bound.
2. Any room can contain threats or power-ups, even the first room the knight enters and the bottom-right room where the princess is imprisoned.

The implementation is done in dungeoneGame.py

Output of dungeonGame.py

```
***=====Dungeon Game=====***

Dungeon : [[-2, -3, 3], [-5, -10, 1], [10, 30, -5]]

---Dynamic Approach---
Steps Taken : ['right', 'right', 'down', 'down']
Minimum HP Required : 7

---Greedy Approach---
Steps Taken : ['right', 'right', 'down', 'down']
Minimum HP Required : 7

Dungeon : [[-2, -3, 3], [-2, -10, 1], [10, 30, -5]]

---Dynamic Approach---
Steps Taken : ['down', 'down', 'right', 'right']
Minimum HP Required : 5

---Greedy Approach---
Steps Taken : ['down', 'down', 'right', 'right']
Minimum HP Required : 5

Dungeon : [[-2, -3, 5], [3, -10, 3], [-10, 2, 2]]

---Dynamic Approach---
Steps Taken : ['right', 'right', 'down', 'down']
Minimum HP Required : 6

---Greedy Approach---
Steps Taken : ['down', 'right', 'right', 'down']
Minimum HP Required : 10
```

Dynamic Approach can be applied because our problem satisfies the 2 conditions for dynamic programming:

1. Overlapping Subproblems
2. Optimal Substructure

**Overlapping Subproblems :** If we take a path that consists of grid  $S_c$ . Let this path be defined as  $P1=S_k+A_1\ldots+S_c+\ldots+A_n+S_q$ . Where  $S_k$  refers to the grid where our knight starts and  $S_q$  refers to the grid where our queen is seated. Then there can exist another path  $P2$  such that  $P2=S_k+G_1\ldots+S_c+\ldots+G_n+S_q$ . Meaning not considering  $S_k$  and  $S_q$  there can exist multiple paths moving between the knight and queen that may share a single or more than a single grid. Hence when we compute these paths for our constraint of minimum HP we will use the solution of each grid for respective paths redundantly. This condition exhibits overlapping sub problems. To reduce the computation overhead we can store it in a table and access it later.

**Optimal Substructure:** Now consider the point where an optimal solution for a sub problem also produces an optimal solution for the problem at hand. Well this holds true as well. If a grid  $x$  lies in our solution of a path between knight and the queen that gives us the optimal solution. Then our problem breaks down into finding an optimal path between source and  $x$  and  $x$  and destination grid. The source and  $x$  along with  $x$  and destination problem can be similarly quantified as a source and destination problem. As when there exist multiple paths between  $u$  and  $v$  grid we need to choose a path such that our constraint of HP remains minimum. What we were doing before between source and destination is the same we need to do between source and  $x$  and  $x$  and destination

### Recursive Solution:

Let our dynamic Table be  $T[][]$  of dimension  $r*c$  where  $r=M$  &  $c=N$  of dungeon

$$T[r][c]=\max(1 - \text{dungeon}[r][c], 1)$$

$$T[i][c] = \max(T[i + 1][c] - \text{dungeon}[i][c - 1], 1) \text{ where } i=\{1,2\ldots r-1\}$$

$$T[r][j] = \max(\text{hp}[r - 1][j] - \text{dungeon}[r - 1][j], 1) \text{ where } j=\{1,2\ldots c-1\}$$

$$\text{hp}[i][j] = \min(\max(T[i][j + 1] - \text{dungeon}[i][j], 1) , \max(T[i + 1][j] - \text{dungeon}[i][j], 1)) \text{ where } 0<i<r \text{ and } 0<j<c$$

The dynamic approach produces the desired optimal result always selecting the path that is best out of all paths. Where the constraint on the path is of selecting minimum health (HP) of our knight so he can safely reach the queen. The dynamic working is that it exhausts all paths and calculates the minimum HP required by knight at starting. This it does by imposing the constraints on our problem while we calculate the dynamic table. From the table, we select the value that is the minimum. The dynamic approach helps because paths from source to destination (knight to queen) intersect and overlap many times. Instead of calculating the same problem of health required for the same or intersected paths we calculate it once and reuse the solution again. Making our computation efficient of the paths.

The greedy approach does not consider all the paths but at given period makes a quick greedy decision between 2 cells right and down. That is which cell selection will benefit its constraint of keeping its HP to minimum. Once it decides this step it selects it and moves forward. Now it again takes a similar decision, but this time it only will do so with respect to present situation. The paths it has left in the past state are not accountable. There is always a good possibility that the paths it left and did not choose in the past maybe those paths in the future were overall fruitful than the one it will select greedily. The 3rd example in the above output clearly explains the point.

Complexity	Dynamic	Greedy
Time	$(r-1)*(c-1) + (r+c-2)$	$(r+c-2)$
Space	$r*c$	2 [constant]

$r=M$  &  $c=N$  of dungeon grid space.

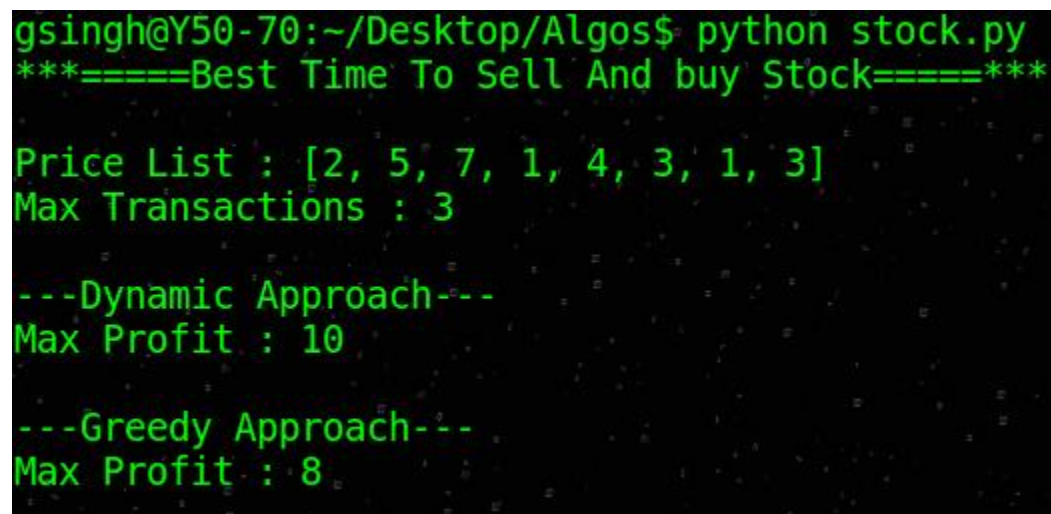
It is clearly visible that space time complexity for greedy is less demanding. Greedy sometimes will give the optimal solution and sometimes not, but its accurateness of selecting an optimal path is also not certain. It's alpha-approximation has no bounds, meaning there is no way we can determine how much worse can the solution be with respect to optimal solution. It may or may not give a solution close to the optimal solution, which will always depend upon the layout of dungeon. Depending upon the usage of the problem an approach among the two can be chosen, a criteria of accurateness vs space time complexity and the size of dungeon will help to determine which approach is the best for that scenario.

## Case Study 2: Best Time To Buy And Sell Stock

### Problem Statement:

Say you have an array for which the  $i^{\text{th}}$  element is the price of a given stock on day  $i$ . Design an algorithm to find the maximum profit. You may complete at most  $k$  transactions. You may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

The implementation is done in stock.py Output of stock.py



```
gsingh@Y50-70:~/Desktop/Algos$ python stock.py
***=====Best Time To Sell And buy Stock=====***

Price List : [2, 5, 7, 1, 4, 3, 1, 3]
Max Transactions : 3

---Dynamic Approach---
Max Profit : 10

---Greedy Approach---
Max Profit : 8
```

Dynamic Approach can be applied because our problem satisfies the 2 conditions for dynamic programming:

1. Overlapping Sub problems
2. Optimal Substructure

**Overlapping Sub problems :** Finding a transaction that maximizes the profit is the goal. Though doing multiple transactions is the desired goal. For given set of days the stock price is listed and  $k$  transactions needs to be done to maximize the profit. Now consider the fact that if only 2 transaction were to take place it may or may not include the solution of 1 transaction that maximizes the profit. Similarly, if we had to calculate for  $p$  transactions it may include the  $p-1$  transactions. To calculate the bigger problem, we calculate the sub problems of 1 to  $p-1$  transactions. So, that the results can be reused. It is very much possible that for  $p$  transactions

it may or may not hold all previous computed  $p-1$  transactions.  $1 < p < k$ . New transaction may be selected/formed from subset of previous transactions.

**Optimal Substructure:** If this property holds then only it makes sense to infer solution of  $k$  transactions from  $k-1$  transactions. Consider this if 1 transaction is present in the optimal solution let it be named as  $optK$ , when max transactions is much greater than 1. Then no better transaction can take place between the 2 days of when the stock was bought and then sold for our  $optK$  transaction, otherwise it would contradict the solution being optimal. So, other transactions must lie outside the 2 given days. So, given a list of prices for the stock with respect to days. That list is broken into 2 halves depending upon the buy day of  $optK$  and selling day of  $optK$ . So, remaining transactions will be selected from the remaining list of day prices of stock. Now this problem is same as before hence we can say our problem follows an optimal substructure.

### Recursive Solution:

$T[i][j] = \max(T[i][j-1], LOCAL)$  the left part is if no transacting on  $j^{th}$  day takes place

$LOCAL = \max(price[j] - price[m] + T[i-1][m])$  for  $m=0,1,2 \dots j-1$  finds the best transaction

$T[i][j]$  is a dynamic table that contains the maximum profit by performing most profitable transactions. The  $i$  represent transactions and  $j$  represent days. Another dynamic array named  $LOCAL$  is there that contains the profit of various transactions that take place with respect to given day( $j$ ) and number of max transactions( $i$ ) that can be performed.

At any given  $i, j$  value transaction and day. Either a transaction takes place or it does not. If transaction does not take place, we simply output the value of profit on same day but with one less transaction. Otherwise if a transaction took place then profit should increase with respect to value that was chosen when no transaction took place. This value is computed by forming a new transaction its profit and profit before the new transaction took place.

The problem statement is very simple to understand. Given a set of prices for a given stock on each day. We need to find such possible transactions that give us the maximum profit. Also, making sure that if buy a stock it has to be sold (complete transaction)

When using the dynamic approach, we break the problem into  $k$  transactions. That means we first try to solve the basic problem of finding maximum profit for a given set of prices for only 1 transaction. Using that solution, we try to find another transaction



that leads to the maximization of profit, at the end now making 2 transactions. We continue to do so until we reach k transactions. Processing the older solution of p-1 transactions and adding just 1 transaction at a certain place such that profit is maximized. ( $1 < p < k+1$ ). At the end, we will do so for k transactions and find our solution.

Since it is clearly mentioned that at most k transactions, we can end up in transactions less than k. It may be possible that after x transactions  $x < k$  the max profit may become saturated. Even increasing x won't lead in maximizing the profit. But this condition is handled and our solution will continue to give the solution for x transaction. Meaning we will remain at the same max profit even if we increase the transaction space. All of this we can do by maintaining two list. A global list (glV) that maintains the total maximum profit for each day (rows) with respect to transaction (columns) allocated. Whereas the local list (llV) maintains the profit for current transaction with respect to the selected current price (price at which stock is bought). To find the maximum profit for each transaction and adding them globally in glV with respect to total transactions allocated for all days is what the dynamic table programming achieves.

The greedy approach is very simple when compared to its counterpart dynamic approach. We just select the stock to buy whose price is lowest among the list of prices from the day the last transaction was completed. And then we sell it as soon as we get any profit from that buy, even if it's just marginally small. We continue to do so repeatedly until we are finished with all days or k transactions. We always make sure that we sell the stock on the last day. Though in my approach I have made a small change we automatically buy the stock at the price of Day 1. From there we continue to apply the greedy approach. This is done generally if Day 1 consists a considerably low price with respect to remaining days.

Complexity	Dynamic	Greedy
Time	(days*transactions)	(days-1)
Space	2*days*transactions	4 [constant]

Considering the problem at hand that we want to maximize the profit at hand. Using dynamic programming is a better approach as with greedy we can get stuck with a very poor solution. Though dynamic main disadvantage is the space complexity of maintaining 2 list. And then computing of those 2 list. Considerably for large price list and a big k (transaction number) memory will be a bigger issue than computation. The

greedy algorithm is very hard to implement as there is no way of justifying where the approach should start from.

## Case Study 3 : Knapsack

### Problem Statement:

Two main kinds of Knapsack Problems: Knapsack has its own weight constraint –  $W$ .

Note : Duplicate items are forbidden.

- 1) 0-1 Knapsack:
  - a)  $N$  items (can be the same weight or different), where weight of each item is less than or equal to  $W$
  - b) Have only one of each item , must leave or take each item
- 2) Fractional Knapsack:
  - a)  $N$  items (can be the same or different)
  - b) Can take fractional part of each item as much as you want

Knapsack.py implementation output, having a knapsack with maximum value

```
***=====0/1 Knapsack=====***
Weights : [1, 3, 4, 5] Corresponding Values : [1, 4, 5, 7]
Maximum weight of knapsack : 7

---Dynamic Approach---
Items Included in knapsack : [3, 4]
Max Value : 9

---Greedy Approach---
Items Included in knapsack : [1, 5]
Max Value : 8

***=====Fractional Knapsack=====***
Weights : [1, 3, 4, 5] Corresponding Values : [1, 4, 5, 7]
Maximum weight of knapsack : 7

---Greedy Approach---
Items Included in knapsack : [3, 5]
Max Value : 9.66666666667

---Dynamic Approach---
Max Value : 9.66666666667
```

## 0-1 Knapsack

Dynamic Approach can be applied because our problem satisfies the 2 conditions for dynamic programming:

1. Overlapping Subproblems
2. Optimal Substructure

**Overlapping Subproblems :** The knapsack must contain items such that the total weight of items is less than or equal to the maximum weight the knapsack can carry. While constructing the solution, we must look at the various possibilities / combinations of items. Either an item will be in the knapsack or it won't be there. In all these combinations, we would be adding some collective items together to find the total weight. These additions of items together are our overlapping sub problems which we can avoid recomputing repeatedly by storing the output.

**Optimal Substructure:** If somehow, we can calculate or find the items that maximize the value for total weight  $p$ . Where  $p$  is less than total weight the knapsack( $W$ ) can carry. Then while finding the solution of  $W$  if the problem breaks down into the sub problem of finding the items for weight  $p$ . We can use the previous optimal solution. Means if an item is in the optimal solution then the remaining weight that can be put in the knapsack is the same problem we are left with when we started finding the solution for the  $W$  knapsack. Hence is the combination of optimal solutions of different weights of items.

### Recursive Solution:

$T[0][i] = \text{values}[0]$  if  $\text{weights}[0] \leq i$

$T[i][j] = 0$  if  $j = 0$

$T[i][j] = \max(T[i-1][j], \text{values}[i] + T[i-1][j - \text{weights}[i]])$  {if  $\text{weights}[i] \leq j$ }

$T[i][j] = T[i-1][j]$  {If item weight is greater than total remaining weight of knapsack}

Where  $T[i][j]$  is a dynamic table  $i$  is the reference to different weights/items( $n$ ) that can be placed in the knapsack. Whereas  $j$  is the reference to max weight ( $W$ ) of knapsack ( $1, 2, \dots, W-1, W$ ). Number of items is  $n$ .

The dynamic table is constructed as follows. We take each item and try to place in different build of knapsacks. Knapsack weights ranging from 1 to  $W$ . As soon as the corresponding item weight is less than equal to the weight the knapsack can carry we try to add the item and see if by adding the item do we get a value better than the value we found when the item was not included but the total weight of knapsack was the same.

The new value is computed as follows it is the new value of item added. (\*) Plus the values of items that can still be added depending upon the space left in knapsack after new item has been added. Here if knapsack total weight left is 0 no new item can be added and thus value is 0. While what the dynamic table does is that it keeps on storing the max values for all combination of items for a given build of knapsack. And when we refer to (\*) the value is accessed from the dynamic table itself and not recomputed again.

The greedy approach is very simple it picks the item with the maximum value. Subtracts the weight of the item with the knapsack total weight. It continues to do so until all items are finished. It does not subtract an item whose weight is bigger than the weight the knapsack can carry. At the end, we get an maximum value. Though the solution may not be optimum. By making a greedy choice it tends to minimize the chance of selecting items that might have less value but when such items are combined intuitively together their value is greater than the greedy solution while staying true to the constraint of item weight less than total weight of knapsack.

Complexity	Dynamic	Greedy
Time	(items*W)	items
Space	(items*(W+1))	1 [constant]

W=maximum weight of knapsack

## Fractional Knapsack

Fractional Knapsack greedy approach always produces the optimal solution.

Proof Of Greedy Approach producing optimal solution:

Assume an optimal solution  $A=\{a_1, a_2, \dots, a_n\}$  to the fractional knapsack problem (F) that does not include the item  $i$ , with the greatest value per weight ( $v/w$ ) ratio of all initial items. Suppose  $a_1$  is the item in the solution  $A$  with the greatest value per weight ratio.

Note that we have:

$$V_i / W_i \geq V_{a_1} / W_{a_1}$$

Now, suppose we remove  $a_1$  from  $A$  and we obtain a solution  $A'$  to a subproblem  $F'$  :

$$A' = A - a_1$$

If we combine the solution  $A'$  with the greedy choice  $i$ , we obtain a greater value solution  $B$ . If  $B$  is better than  $A$ , then this is a contradiction, since we assumed  $A$  to be an optimal solution. And thus  $i$  must be included if  $B=A$ , then we have shown that the greedy choice is included anyway, since that would mean that  $V_i / W_i = V_{a1} / W_{a1}$

Thus this shows that the problem follows optimal substructure but the most important thing is that it does not completely follow overlapping subproblem. Means once a problem is solved it is used and consumed only by the next subproblem. Dynamic programming is expensive when compared to greedy algorithm since we need to compute and store results of previous subproblems. Since we have proved that greedy gives an optimal solution there is no need to discuss dynamic approach. As solving through dynamic would be useless. However the dynamic programming code implementation for fractional knapsack is done for the report.

In the greedy approach we just keep on selecting the items with maximum value to weight ratio. And when knapsack weight is less than the weight of the item a portion of that item is inserted into the knapsack and its corresponding value is added to find the maximum value of the knapsack.

Complexity	Dynamic	Greedy
Time	(items*W)	items
Space	(items*(W+1))	items

## CONCLUSION

### **Dungeon Game:**

Dynamic Programming produces a more optimal solution than greedy approach. The constraint of selecting a minimum HP is what makes Dynamic Programming better than greedy. If just a path had to be found that even greedy also can do but the cost at which it does is sometimes expensive than dynamic approach. So Dynamic Wins.

### **Best Time To Buy And Sell Stock:**

There is no sure technique of applying greedy algorithm as we can not decide a greedy starting point. Whereas dynamic though at a high space time complexity delivers the optimal solution. Dynamic is only feasible

### **0-1 Knapsack:**

Applying Greedy approach may lead to that some unwanted space is left in the knapsack and we also haven't reached the maximum value for the knapsack. Whereas dynamic makes sure that it avoids both drawbacks and gives the optimal solution. Dynamic is preferred.

### **Fractional Knapsack:**

Since we proved that greedy always gives optimal solution. And also, that dynamic will always be worse in space and time complexity. Greedy is the preferred choice.

As conclusion, we can state that if greedy gives an optimal solution always we would always prefer it due to is better space time complexity compared to dynamic approach. But if greedy sometimes gives an optimal solution then we must decide how much relaxation we can give on the optimal solution. Basically, a trade-off between complexity and optimal solution. Sometimes greedy won't work at all and there we would require to apply dynamic programming.

# TRAVELLING SALESMAN PROBLEM

It is attached as a separate file named  
**TravellingSalesmanProblem\_Gurleen\_Sudhanshu.pdf**

## REFERENCES

1. [https://en.wikipedia.org/wiki/Dynamic\\_programming](https://en.wikipedia.org/wiki/Dynamic_programming)
2. [https://en.wikipedia.org/wiki/Greedy\\_algorithm](https://en.wikipedia.org/wiki/Greedy_algorithm)
3. [https://en.wikipedia.org/wiki/Held%E2%80%93Karp\\_algorithm](https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm)
4. <http://www.geeksforgeeks.org/dynamic-programming-set-1/>
5. [https://www.youtube.com/watch?v=8LusJS5-AGo&list=PLrmLmBdmIlpsHaNTPP\\_jHHDx\\_os9ItYXr](https://www.youtube.com/watch?v=8LusJS5-AGo&list=PLrmLmBdmIlpsHaNTPP_jHHDx_os9ItYXr)